

Lehrstuhl für
Rechnerarchitektur & Parallele Systeme
Prof. Dr. Martin Schulz
Dominic Prinz
Jakob Schäffeler

Lehrstuhl für
Design Automation
Prof. Dr.-Ing. Robert Wille
Stefan Engels

Einführung in die Rechnerarchitektur

Wintersemester 2024/2025

Übungsblatt 4: RISC-V Teil 3 - Rekursion und Calling Convention

11.11.2024 – 15.11.2024

Bitte beachten Sie, dass ab sofort die **Calling Convention** stets eingehalten werden muss. Details entnehmen Sie bitte der Vorlesung sowie der 4. Zentralübung.

1 Calling Convention

a) Was ist eine Calling Convention?

- „Vertrag“ zwischen Entwicklern auf einem System
- Ziel: Standardisierung der Funktionsweise von Programmen (z.B. Parameterübergabe)
- Konvention wird nicht überprüft, aber System setzt davon aus, dass sie eingehalten wird.

b) Unterteilen Sie die Register in Kategorien und klassifizieren Sie diese nach ihrem Zweck und anderen Unterschieden in der Calling Convention.

Register	Zweck
t0-t6	Temporäre Werte
a0-a7	Parameter / Rückgabewert
sp	Stack pointer
ra	Return Address
s0-s11	persistente Werte
x0/zero	Immer 0

c) Wie können Sie zum Beispiel das Register s0 trotzdem verwenden?

- ① Wert auf Stack sichern
- ② Vor return Wert wiederherstellen

- d) Sie schreiben ein Programm in Assembly und rufen ein Unterprogramm auf. Welche Register sind danach garantiert unverändert? Welche müssen Sie vorher sichern um zu garantieren, dass das Programm nicht abstürzt?

Unverändert: S-Register, sp

Zu sichern: f-Register, a-Register, VA

- e) Warum benötigt man eine Calling Convention? Was sind die Vorteile?

- Nicht alle Register müssen bei jedem Unterprogrammaufruf gesichert werden
- Code lesenbar + einfach erweiterbar

- f) Wie werden die Parameter und Rückgabewerte in den folgenden Funktionen nach Calling Convention übergeben/zurückgegeben?

- a) `char add_char(char a, char b)`
- b) `uint64_t add(uint64_t a, uint64_t b)`
- c) `uint64_t add(uint64_t a, uint64_t b)`
- d) `uint128_t copy_and_increment(uint64_t a)`

- a) $a \in a0$ $b \in a1$ yet $\in a0$

- b) a in a_1, a_0 b in a_3, a_2 ret in a_1, a_0

- c) a in a_0 b in a_1, a_1 ret in a_1, a_0

- b) a in a2,a1 vet in a0
 ↑
 Pointer / Adresse

2 Rekursion in der Theorie

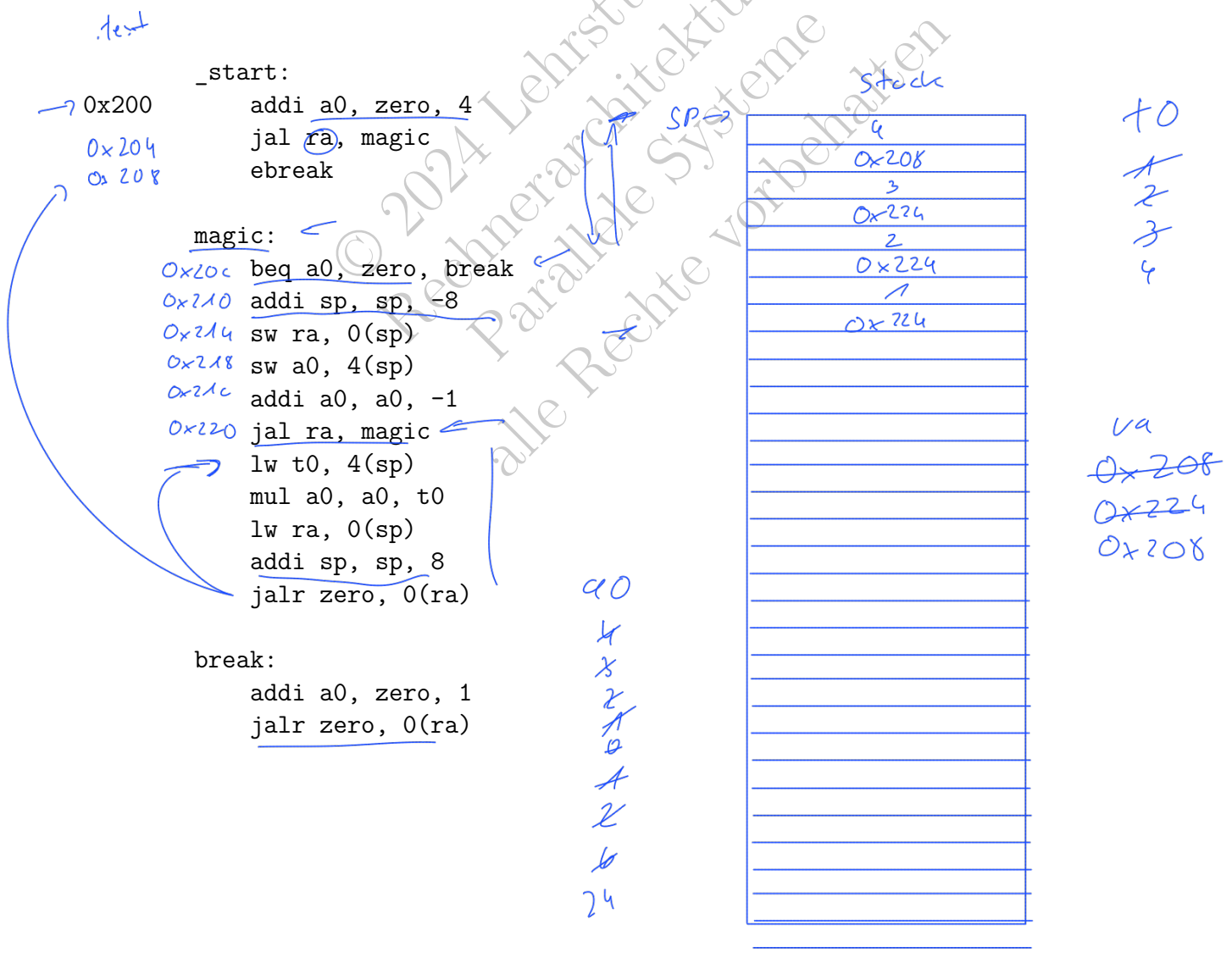
Gegeben sei folgendes Programm, das ab der Startadresse 0x200 im Speicher liegt. Das heißt die Bitfolge der Instruktion `addi a0, zero, 4` steht an 0x200.

- Annotieren Sie alle Zeilen mit den entsprechenden Adressen der Instruktionen. Sie dürfen davon ausgehen, dass sie aufeinanderfolgend im Speicher liegen und keine komprimierten Instruktionen verwendet werden.
- Gehen Sie Schritt für Schritt durch die ausgeführten Befehle. Notieren Sie sich jeden Schritt, in dem sich der Stack ändert, sowie dessen Zustand zu diesem Zeitpunkt.
- Was berechnet das Unterprogramm `magic`?

Fakultät von 4

- d) Hält das Programm die Calling Convention ein? Wenn nein, was müsste man ändern?

X Stack nicht 16-Byte aligned



3 Größter gemeinsamer Teiler

Erstellen Sie ein Unterprogramm `ggT`, welches den größten gemeinsamen Teiler von zwei in `a0` und `a1` übergebenen Zahlen rekursiv berechnet. Das Ergebnis soll in `a0` zurückgegeben werden. Als Hilfestellung ist unten C-Code zur rekursiven Berechnung des `ggT` gegeben. Achten Sie auch darauf die Calling Convention einzuhalten.

Hinweis: `unsigned` steht für `unsigned int`.

```
unsigned ggT(unsigned a, unsigned b) {  
    if (a==b)  
        return a;  
    else if (a < b)  
        return ggT(a, b-a);  
    else  
        return ggT(a-b, b);  
}
```

4 optional: Rekursive Folge

Schreiben Sie ein Unterprogramm, welches diese Folge rekursiv berechnet:

$$a_n = 2 \cdot a_{n-1} + n, \quad a_0 = 10$$

Verwenden Sie nur Befehle aus Tabelle 1.

Hinweis: Sie können das Ergebnis mithilfe dieser Formel überprüfen: $a_n = -n + 3 \cdot 2^{n+2} - 2$.

5 Tribonacci (Hausaufgabe 04)

Bearbeitung und Abgabe der Hausaufgabe 04 auf <https://artemis.in.tum.de/courses/401> bis **Sonntag, den 17.11.2024, 23:59 Uhr**.

Referenzmaterial

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address] _{7:0})
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address] _{15:0})
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address] _{31:0}
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address] _{7:0})
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address] _{15:0})
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srlr rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srair rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address] _{7:0} = rs2 _{7:0}
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address] _{15:0} = rs2 _{15:0}
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address] _{31:0} = rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 – rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2 _{4:0}
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2 _{4:0}
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2 _{4:0}
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

* Encoded in instr_{31:25}, the upper seven bits of the immediate field

Abbildung 1: RISC-V 32-Bit Integerbefehle